PROCESSES AND OPERATING SYSTEMS

FIU

FLORIDA INTERNATIONAL UNIVERSITY

PROCESSES

- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - Registers;
  - Memory;
  - Open files, etc.
- The operating system manages processes.

TERMS

- Thread = lightweight process
  - The entity within a process that can share many system resources with others.
    - Address space, executable code, global variables, etc.
      - How about stack ?
  - Each process has at least one thread, i.e., primary thread
  - Faster context switching among threads than processes
- Reentrancy
  - a single copy of the program's instructions in memory can be ***safely*** shared by multiple, separate users, object classes, or processes

EXAMPLE OF NON-REENTRANCY

```
int var = 1;

int f( ) {
var = var + 2;
return var; }

int g( )
{
return f() + 2;
}
```

EXAMPLE OF REENTRANCY

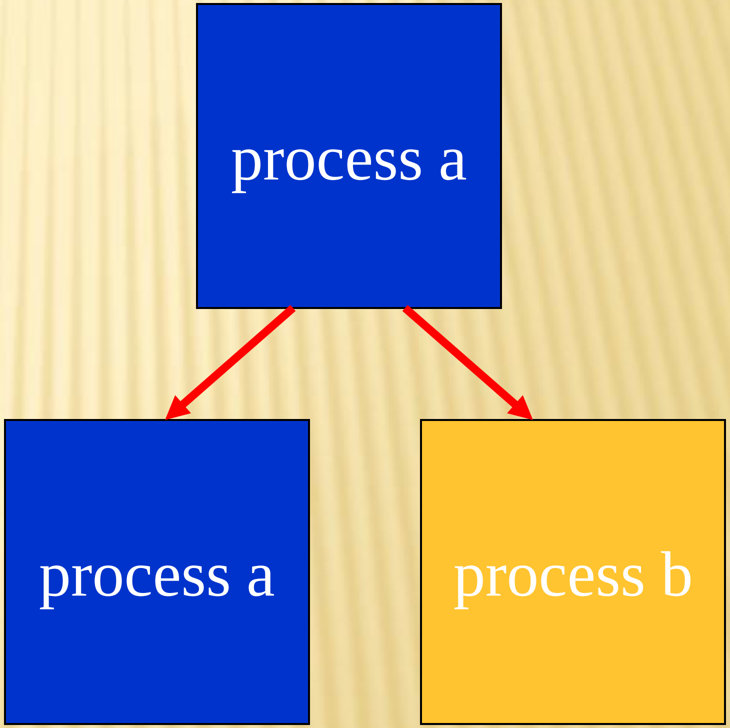
```
int f(int var) {
var = var + 2;
return var; }

int g(int var)
{
return f(var) + 2;
}
```

MULTIPLE TASKING

- Create a process
- Context switching
- Process State and Scheduling
- Interprocess communication
- Real-time operating system (RTOS)

CREATE PROCESSES IN POSIX

- Create a process with fork:
  - Exact copy for parent and child except for the return value of fork().



FORK()

- The fork process creates child:

```
childid = fork();
if (childid == 0) {
    /* child operations */
} else {
    /* parent operations */
}
```

EXECV()

- Overlays child code:

```
childid = fork();
if (childid == 0) {
    execv(“mychild”,childargs);
    perror(“execv”);
    exit(1);
}
```

file with child code

MULTIPLE TASKING

- Create a process
- Context switching
- Process State and Scheduling
- Interprocess communication
- Real-time operating system (RTOS)

CONTEXT SWITCHING

- How
  - Copy all context (registers), keeping proper return value for PC.
  - Copy new context into CPU state.
- Who in control of context switching

CONTEXT SWITCHING IN ARM

- Save old process:

```
STMIA    r13,{r0-r13}^
MRS      r0,SPSR
STMDB  r13,{r0, r15}
```

; r14: contains the next instruction after return from sub-procedure

; r15: program counter (pc)

- Start new process:

```
ADR        r0,NEXTPROC
LDR        r13,[r0]
LDMDB r13,{r0, r14}
MSR       SPSR,r0
LDMIA  r13,{r0-r13}^
MOV   pc, r14
```

David Jaggar, e.d., *Advanced RISC Machines Architectural Reference Manual*, London: Prentice Hall, 1995.

FIU
Florida International University

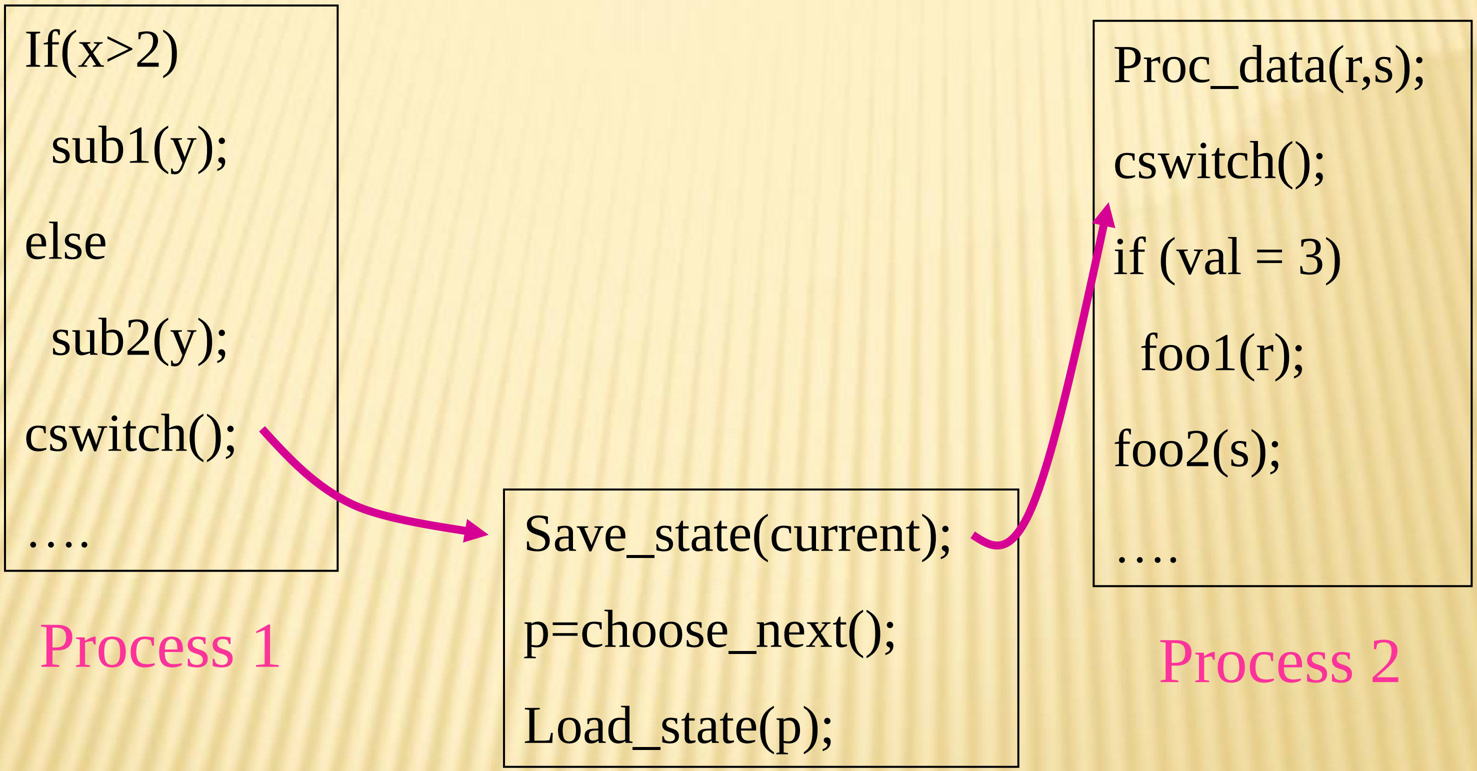
CONTEXT SWITCHING

- How
  - Copy all context (registers), keeping proper return value for PC.
  - Copy new context into CPU state.
- Who is in control of context switching
  - Co-operative multitasking
  - Preemptive multitasking
  - Co-routine

CO-OPERATIVE MULTITASKING

- What
    - One process gives up the CPU to another voluntarily
    - Each process allows a context switch at cswitch() call.
    - Separate scheduler chooses which process runs next.

COOPERATIVELY MULTITASKING EXAMPLE



```
If(x>2)
  sub1(y);
else
  sub2(y);
cswitch();
….
```

Process 1

```
Save_state(current);
p=choose_next();
Load_state(p);
```

Scheduler

```
Proc_data(r,s);
cswitch();
if (val = 3)
  foo1(r);
foo2(s);
….
```

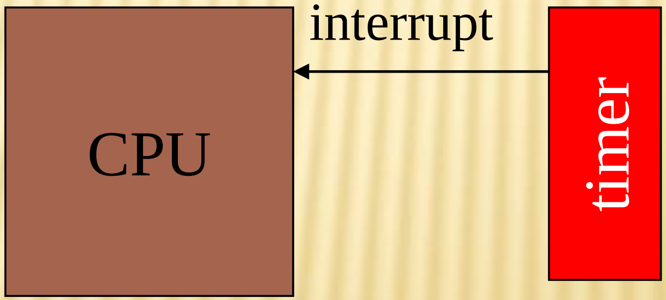
Process 2

CO-OPERATIVE MULTITASKING

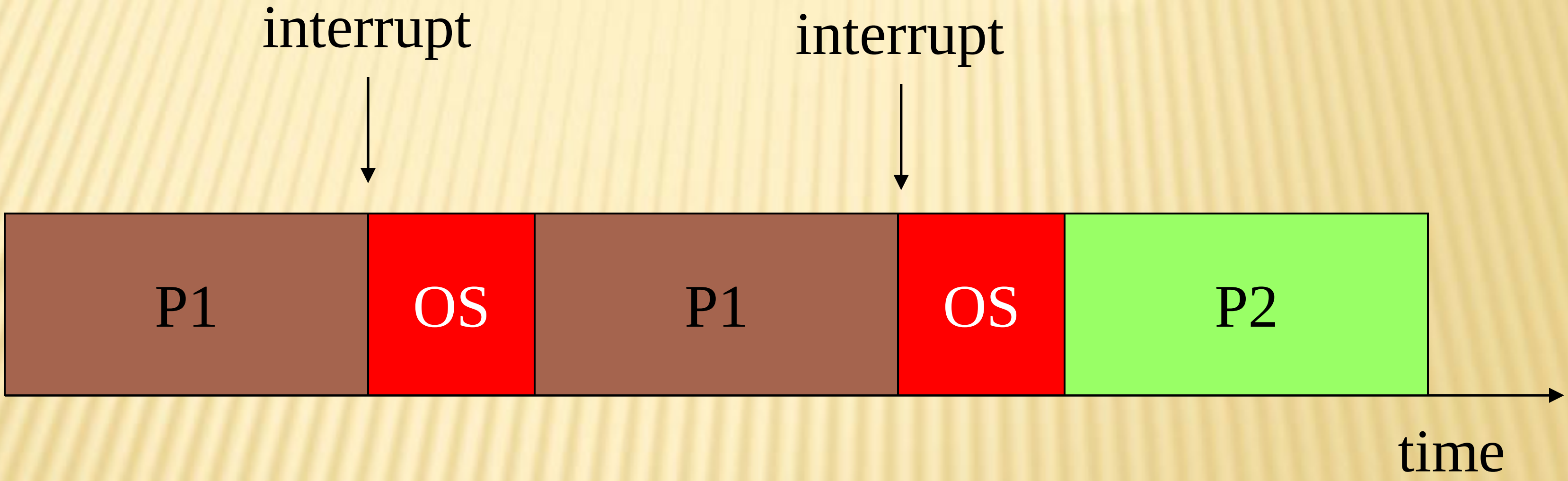
- Hides context switching mechanism;
- Relies on processes to give up CPU.
- Programming errors can keep other processes out:
  - process never gives up CPU;
  - process waits too long to switch, missing input.

PREEMPTIVE MULTITASKING

- OS controls when contexts switches and determines what process runs next.
- Interrupts (by timer, external events) cause OS to switch contexts:



FLOW OF CONTROL WITH PREEMPTION



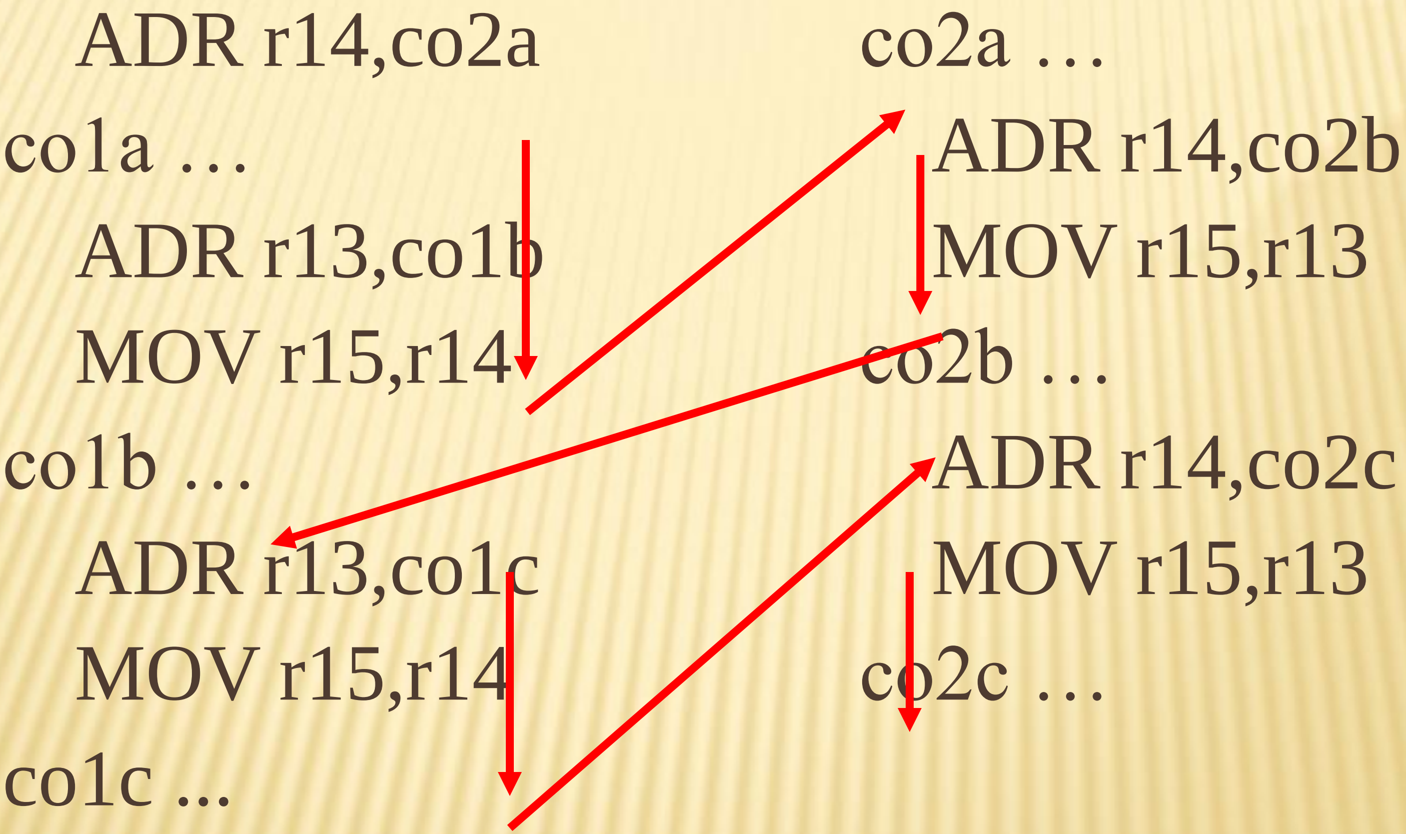
PREEMPTIVE CONTEXT SWITCHING

- Interrupt gives control to OS, which saves interrupted process's state in an activation record.
- OS chooses next process to run.
- OS installs desired context as current CPU state.

CO-ROUTINE FOR MULTIPLE TASKING

- Rooted in assembly programming
- Rarely used today
- Generalize subroutines to allow multiple entry points and suspending and resuming of execution at certain locations
- An example

CO-ROUTINES



```
    ADR r14,co2a
co1a …
    ADR r13,co1b
    MOV r15,r14
co1b …
    ADR r13,co1c
    MOV r15,r14
co1c ...
```

```
co2a …
    ADR r14,co2b
    MOV r15,r13
co2b …
    ADR r14,co2c
    MOV r15,r13
co2c …
```

Co-routine 1

Co-routine 2

r15: the program counter register

MULTITASKING WITH CO-ROUTINE

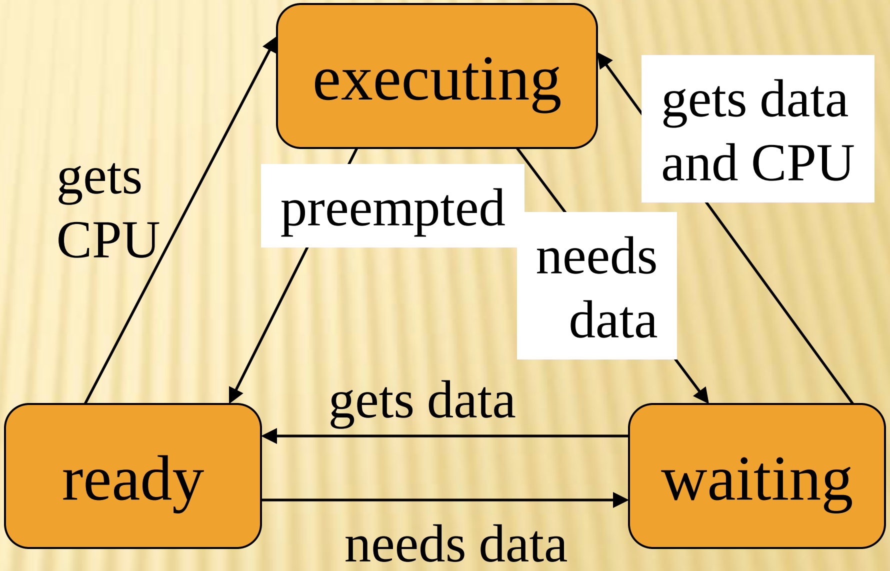
- Like subroutine, but caller determines the return address.
- Co-routines voluntarily give up control to other co-routines.
- Pattern of control transfers is embedded in the code.

MULTIPLE PROCESS

- Create a process
- Context switching
- Process State and Scheduling
- Interprocess communication
- Real-time operating system (RTOS)

PROCESS STATE

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.



SCHEDULING

- The CPU is often shared among several processes.
  - Cost.
  - Energy/power.
  - Physical constraints.
- Someone must be responsible for giving the CPU to processes.
  - Co-operation between processes.
  - RTOS.

EMBEDDED VS. GENERAL-PURPOSE SCHEDULING

- Workstations try to improve the throughput and fairness CPU access.
- Embedded systems must meet deadlines and other constraints.
  - Low-priority processes may not run for a long time.

TIMING REQUIREMENTS ON PROCESSES

- Period: interval between process activations.
- Rate: reciprocal of period.
- Initiation time: time at which process becomes ready.
- Deadline: time at which process must finish.
- Execution time: execution time without preemption

SCHEDULING METRICS

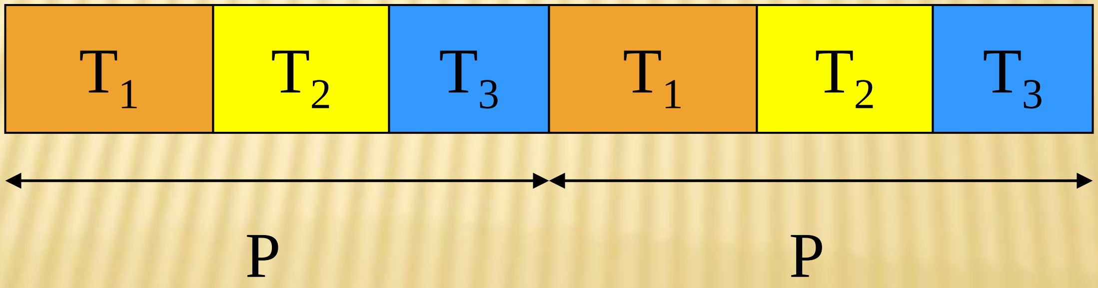
- CPU utilization:
  - Fraction of the CPU that is doing useful work.
  - Often calculated assuming no scheduling overhead.
  - Utilization:
    - $U = [\ \Sigma_{t1 \leq t \leq t2}\ T(t)\ ] / [t2 - t1]$
      - T(t): *useful* execution time.
- Response time
  - Time from when the task is ready to the task being finished

SCHEDULING METHODS

- Cyclic scheduling
- Round robin scheduling
- Preemptive scheduling

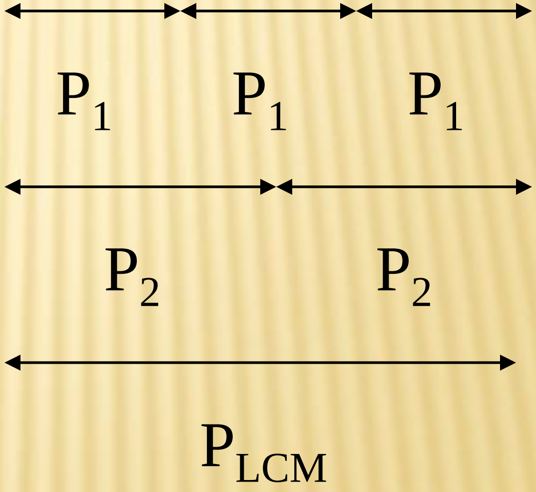
CYCLIC SCHEDULING

- Schedule task according pre-determined schedule
- Schedule in time slots.
  - Same process activation irrespective of workload.
- Time slots may be equal size or unequal.



THE ASSUMPTIONS

- Trivial scheduler -> very small scheduling overhead.
- Can’t handle unexpected loads.
  - Must schedule a time slot for aperiodic events.
- Schedule based on the hyperperiod of the process periods.



P_1 P_1 P_1

P_2 P_2

P_{LCM}

HYPERPERIOD

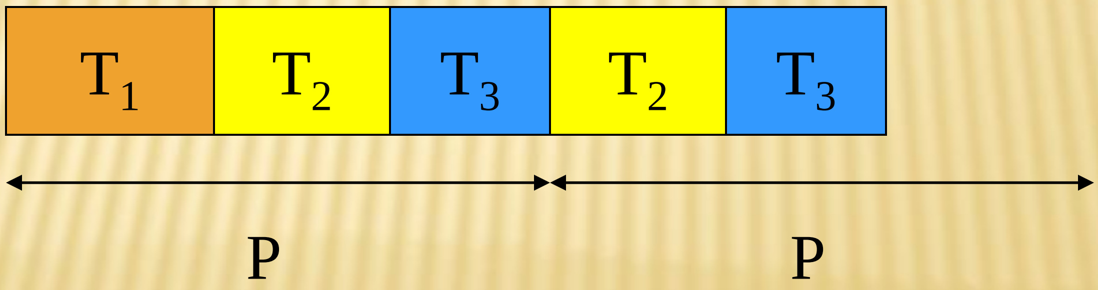
- Hyperperiod: least common multiple (LCM) of the task periods.
- Hyperperiod can be very long if task periods are not chosen carefully.
  - Larger scheduling table
  - More scheduling overhead

HYPERPERIOD EXAMPLE

- Long hyperperiod:
  - P1 7 ms.
  - P2 11 ms.
  - P3 15 ms.
  - LCM = 1155 ms.
- Shorter hyperperiod:
  - P1 8 ms.
  - P2 12 ms.
  - P3 16 ms.
  - LCM = 96 ms.

ROUND-ROBIN

- Schedule process only if ready.
  - Always test processes in the same order.
- Variations:
  - Constant/weighted time slots
  - Start round-robin again after finishing a round.
- Better adaptivity
  - Can be adapted to handle unexpected load.



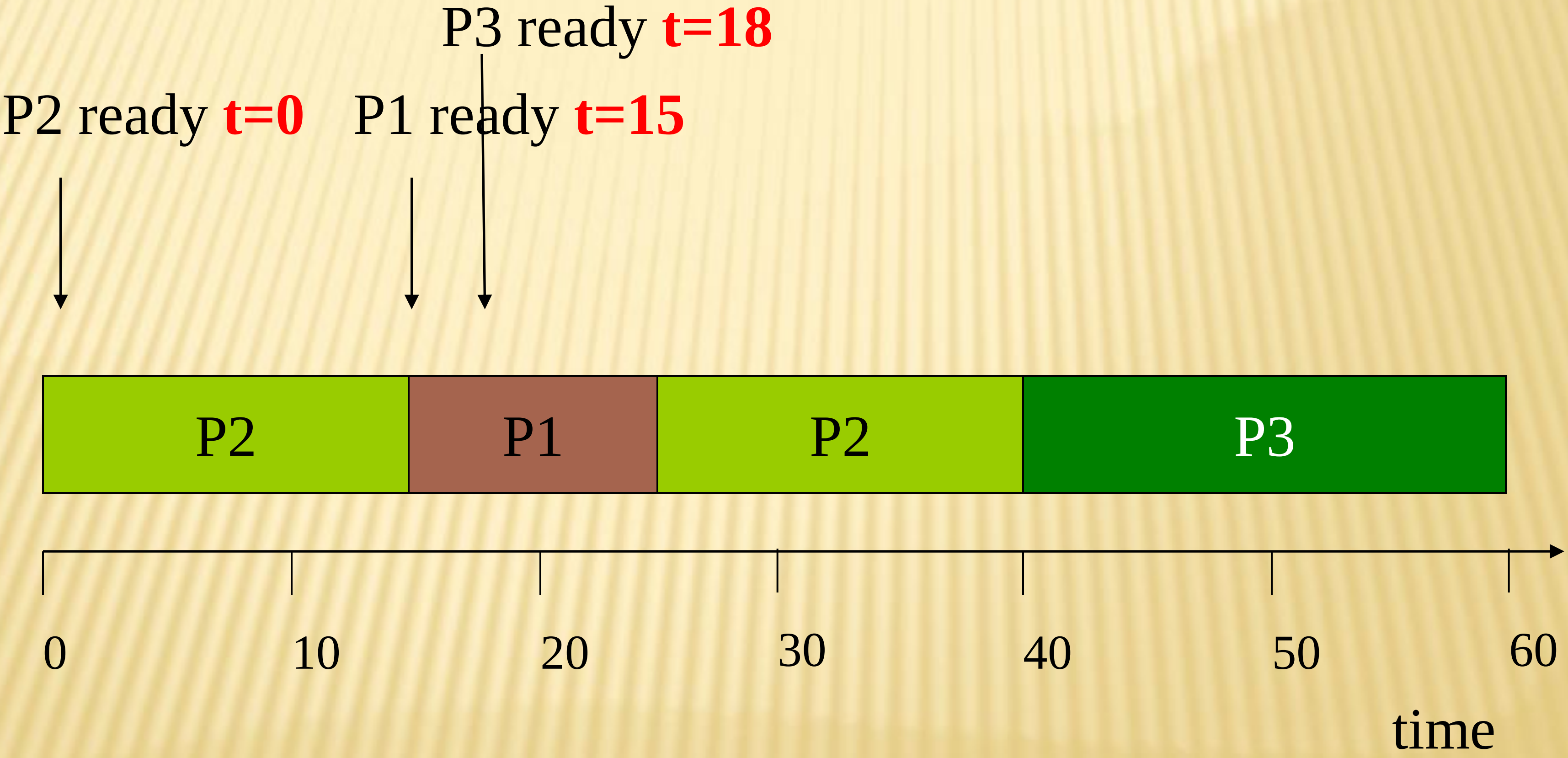
PRIORITY-DRIVEN SCHEDULING

- Each process has a priority.
- CPU runs the highest-priority process that is ready.
- Priorities determine scheduling policy:
  - fixed priority;
  - time-varying priorities.

PRIORITY-DRIVEN SCHEDULING EXAMPLE

- Rules:
  - each process has a fixed priority (1 highest);
  - highest-priority ready process gets CPU;
  - process continues until done.
- Processes
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

PRIORITY-DRIVEN SCHEDULING EXAMPLE



P3 ready **t=18**

P2 ready **t=0**

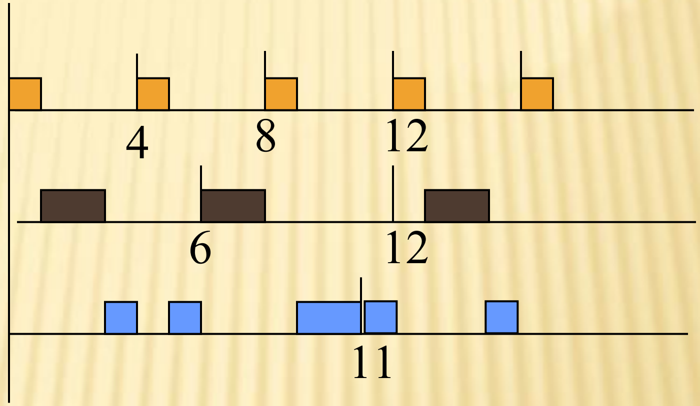
P1 ready **t=15**

TWO PRIORITY-BASED PREEMPTIVE SCHEDULING

- Rate Monotonic Scheduling (RMS)
  - Shortest-period process gets highest priority, i.e. priority inversely proportional to period;
    - Higher the rate (smaller the period), higher the priority
  - Schedulability analysis
- Earliest Deadline First (EDF)
  - Process closest to its (absolute) deadline has highest priority.
  - Schedulability analysis

RMS EXAMPLE

P1=D1=4 C1=1
P2=D2=6 C2=2
P3=D3=11 D3=4



RMS SCHEDULABILITY ANALYSIS

- Can all tasks meet their deadlines?
  - A simple RMS model
    - All processes are periodic (with period Pi) and run on a single CPU.
    - Process execution time (Ci) is constant (worst case).
    - Deadline is at end of period (Di=Pi).
    - Zero context switch time.
  - Utilization bound analysis

  - Worst Case Response Time Analysis
    - If the longest response time is less than the deadline, it is schedulable
    - When a task will have the longest response time
      - Critical instant: scheduling state that gives worst response time.
      - Critical instant occurs when all higher-priority processes are ready to execute simultaneously.

UTILIZATION BOUND

- Utilization factor

$$U = \sum_i \frac{C_i}{P_i}$$

- Theorem: For a set of m tasks with fixed priority order, the least upper bound to processor utilization is

$$U_b = m(2^{1/m} - 1)$$

- In another word, for a given task set, if the utilization factor is no more than the corresponding bound, then the task set is schedulable, i.e., all tasks can meet their deadlines.
  - E.g. m=2, U_b=0.83; m= 3, U_b=0.78; for large m, $U_b \rightarrow \ln 2 = 0.69$

UTILIZATION BOUND (CONT'D)

- A sufficient condition
  - Many feasible task can have higher utilization

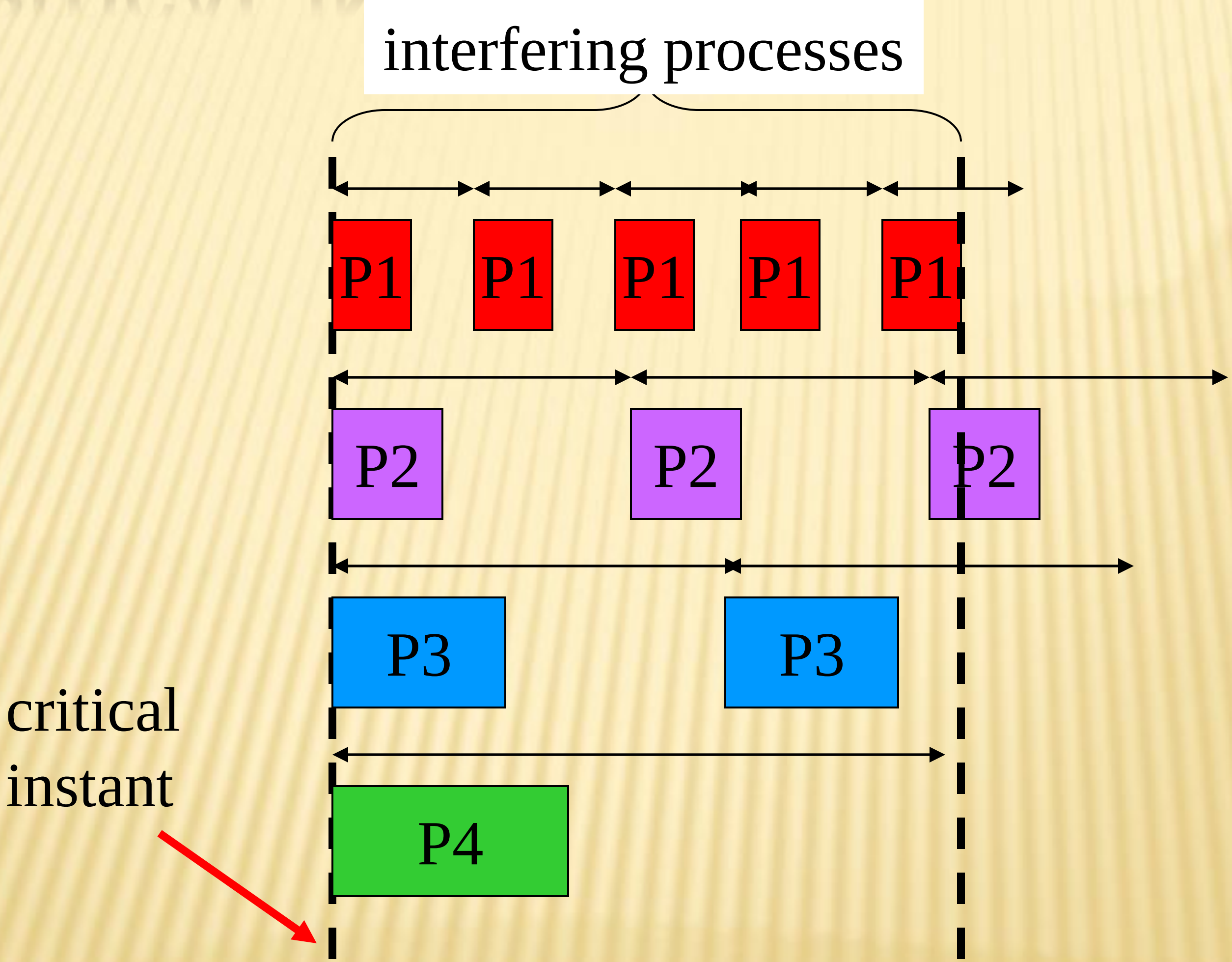
- Many feasible fixed-priority task sets cannot 100% utilize the processor

RMS SCHEDULABILITY ANALYSIS

- Can all tasks meet their deadlines?
  - A simple RMA model
    - All processes are periodic (with period Pi) and run on a single CPU.
    - Process execution time (Ci) is constant (worst case).
    - Deadline is at end of period (Di=Pi).
    - Zero context switch time.
  - Utilization bound analysis

  - Worst Case Response Time Analysis
    - If the longest response time is less than the deadline, it is schedulable
    - When a task has the longest response time
      - Critical instant: scheduling state that gives worst response time.
      - Critical instant occurs when all higher-priority processes are ready to execute simultaneously.

CRITICAL INSTANT



WORST CASE RESPONSE TIME ANALYSIS

- Mathematic formulation of the worst case response time for each task is possible
  - For more details, see the following reference
    - Lehoczky, J.; Sha, L.; Ding, Y. (1989), "The rate monotonic scheduling algorithm: exact characterization and average case behavior", *IEEE Real-Time Systems Symposium*, pp. 166–171
- Key points of RMS
  - A fixed priority scheduling method
  - The optimal (fixed) priority assignment
    - If a task set is schedulable with any other fixed priority assignment, it is schedulable with RMS.
  - The worst case response time of a task occurs when it starts at the same time when all higher priority tasks start

EXAMPLES

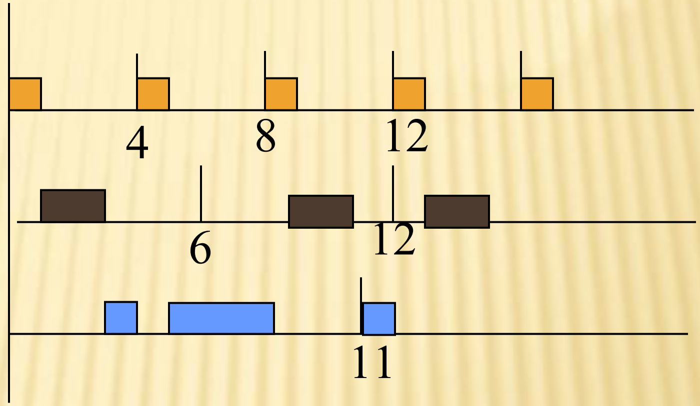
- Example 1: A task set contains three tasks. Let
  - P1=D1=100, P2=D2=150, P3=D3=300
  - C1=40, C2=40, C3=20
  - ***Since*** $U = 40/100 + 40/150 + 20/300 = 0.733 < 3 (2^{1/3} - 1) = 0.78$
  - ***The task set is schedulable***
- Example 2: A task set contains two tasks. Let
  - P1=D1=100, P2=D2=200
  - C1=50, C2=100
  - $U = 50/100 + 100/200 = 1.0 > 2 (2^{1/2} - 1) = 0.83$
    - ***Cannot be sure if the task set is schedulable or not***
  - *It is in fact schedulable according to the worst case response time analysis*
    - ***Since there is no task with higher priority, its longest response time is*** $50 <= D1$
    - ***The longest response time for task 2 is the response time of its first job (the critical instant since all tasks start at the same time t=0). Its response time is (if you draw the timing diagram)*** $200 <= D2$.

TWO PRIORITY-BASED PREEMPTIVE SCHEDULING

- Rate Monotonic Scheduling (RMS)
  - Shortest-period process gets highest priority, i.e. priority inversely proportional to period;
  - Schedulability analysis
- Earliest Deadline First (EDF)
  - Process closest to its (absolute) deadline has highest priority.
  - Schedulability analysis

EDF EXAMPLE

P1=D1=4 C1=1
P2=D2=6 C2=2
P3=D3=11 D3=4



EARLIEST-DEADLINE-FIRST SCHEDULING

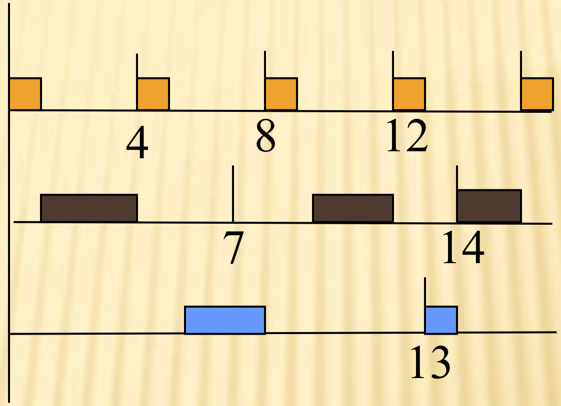
- EDF
  - dynamic priority scheduling scheme.
  - Requires recalculating processes at every timer interrupt.
- Schedulability analysis
  - Theorem: A given task set is feasible by EDF if and only if the total utilization factor U <=1, i.e.

$$U = \sum_i \frac{C_i}{P_i} \leq 1$$

  - Can fully utilize the processor

EXAMPLE

T1=D1=4 C1=1
T2=D2=7 C2=3
T3=D3=13 C3=3



Since U= 1/4 + 3/7 + 3/13 = 0.91 < 1, therefore, the above task set is schedulable.

PRIORITY INVERSION

- Priority inversion: low-priority process keeps high-priority process from running.
- Improper use of system resources can cause scheduling problems:
  - Low-priority process grabs I/O device.
  - High-priority device needs I/O device, but can’t get it until low-priority process is done.
- Can cause deadlock.
  - Deadlock: two or more processes are waiting for each other to finish but neither can do.

MULTIPLE PROCESS

- Create a process
- Context switching
- Process State and Scheduling
- Interprocess communication
- Real-time operating system (RTOS)

INTERPROCESS COMMUNICATION

- Interprocess communication (IPC): OS provides mechanisms so that processes can pass data.
- Two schemes
  - Shared memory:
    - processes have some memory in common;
    - must cooperate to avoid destroying/missing messages.
  - Message passing:
    - processes send messages along a communication channel, i.e. message queue
    - no common address space.

RACE CONDITION IN SHARED MEMORY

- Race condition
  - Output dependent on the sequence of events
  - Example
    - Event 1: CPU 1 reads flag.
    - Event 2: CPU 2 reads flag.
    - Event 3: CPU 1 sets flag to one.
    - Event 4: CPU 2 sets flag to two.
- The producer/consumer problem

EXAMPLE: PRODUCER/CONSUMER

- Share *buffer*[*N*], *count*
  - *count* = # of valid data items in *buffer*
- *processA* produces data items and stores in *buffer*
  - If *buffer* is full, must wait
- *processB* consumes data items from *buffer*
  - If *buffer* is empty, must wait
- Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs. Say "count" is 3.
  - *A* loads *count* (*count* = 3) from memory into register R1 (R1 = 3)
  - *A* increments R1 (R1 = 4)
  - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *A* stores R1 back to *count* in memory (*count* = 4)
  - *B* stores R2 back to *count* in memory (*count* = 2)
  - *count* now has incorrect value of 2

```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N );/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

MUTUAL EXCLUSION

- Certain sections of code should not be performed concurrently
  - Critical section
    - Possibly noncontiguous section of code where simultaneous updates, by multiple processes to a shared memory location, can occur
- When a process enters the critical section, other processes must be locked out until it leaves the critical section
  - Mutex
    - A shared object used for locking and unlocking segment of shared data
    - Disallows read/write access to memory it guards
    - Multiple processes can perform lock operation simultaneously, but only one process will acquire lock
    - All other processes trying to obtain lock will be put in blocked state until unlock operation performed by acquiring process when it exits critical section
    - These processes will then be placed in runnable state and will compete for lock again

USING MUTEX FOR THE CONSUMER-PRODUCER PROBLEM

- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other
- Following the same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be put in blocked state
  - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *B* stores R2 back to *count* in memory (*count* = 2)
  - *B* executes *unlock* operation
    - *A* is placed in runnable state again
  - *A* loads *count* (*count* = 2) from memory into register R1 (R1 = 2)
  - *A* increments R1 (R1 = 3)
  - *A* stores R1 back to *count* in memory (*count* = 3)
- *Count* now has correct value of 3
- ***Problems?***

```
data_type buffer[N];
int count = 0;
mutex count_mutex;
void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    count_mutex.lock();
    while( count == N );/*loop*/
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
    count_mutex.unlock();
  }
}
void processB() {
  int i;
  while( 1 ) {
    count_mutex.lock();
    while( count == 0 );/*loop*/
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    count_mutex.unlock();
    consume(&data);
  }
}
void main() {
  create_process(processA);
  create_process(processB);
}
```

CONDITION VARIABLES

- Condition variable is an object that has 2 operations, signal and wait
- When process performs a wait on a condition variable, the process is blocked until another process performs a signal on the same condition variable
- How is this done?
  - Process *A* acquires lock on a mutex
  - Process *A* performs wait, passing this mutex
    - Causes mutex to be unlocked
  - Process *B* can now acquire lock on same mutex
  - Process *B* enters critical section
    - Computes some value and/or make condition true
  - Process *B* performs signal when condition true
    - Causes process *A* to implicitly reacquire mutex lock
    - Process *A* becomes runnable

CONDITION VARIABLE EXAMPLE: CONSUMER-PRODUCER

- 2 condition variables
  - *buffer_empty*
    - Signals at least 1 free location available in *buffer*
  - *buffer_full*
    - Signals at least 1 valid data item in *buffer*
- *processA*:
  - produces data item
  - acquires lock (*cs_mutex*) for critical section
  - checks value of *count*
  - if *count* = *N*, *buffer* is full
    - performs wait operation on *buffer_empty*
    - this releases the lock on *cs_mutex* allowing *processB* to enter critical section, consume data item and free location in *buffer*
    - *processB* then performs signal
  - if *count* < *N*, *buffer* is not full
    - *processA* inserts data into *buffer*
    - increments *count*
    - signals *processB* making it runnable if it has performed a wait operation on *buffer_full*

Consumer-producer using condition variables

```
data_type buffer[N];
int count = 0;
mutex cs_mutex;
condition buffer_empty, buffer_full;
void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    cs_mutex.lock();
    if( count == N ) buffer_empty.wait(cs_mutex);
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
    cs_mutex.unlock();
    buffer_full.signal();
  }
}
void processB() {
  int i;
  while( 1 ) {
    cs_mutex.lock();
    if( count == 0 ) buffer_full.wait(cs_mutex);
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    cs_mutex.unlock();
    buffer_empty.signal();
    consume(&data);
  }
}
void main() {
  create_process(processA); create_process(processB);
}
```

SEMAPHORE VS MUTEX

- Mutex
  - Lock/unlock operation
  - At any time, only one process can enter the critical section
    - A bathroom with one stall
- Semaphore
  - A semaphore has a non-negative integer value (S >=0)
  - Wait/post operation (atomic operation, i.e. only one operation can be executed at one time)
    - Wait (DOWN)
      - Decrease semaphore value by 1. If S = 0, blocks.
    - Post (UP)
      - Increase semaphore value by 1.
  - Multiple processes can enter a critical section concurrently
    - A bathroom with multiple stalls
  - Mutex is a binary semaphore (max S = 1)

USING SEMAPHORES FOR CONSUMER-PRODUCER PROBLEM

- Mutex is similar to a binary semaphore
- *processA*:
  - produces data item
  - If the buffer is not full (empty > 0) and is allowed to access the critical section (cs_sem>0)
  - Increments *count*
  - exit *critical section*
  - *Signal processes waiting on due to the empty buffer*
- *processB*:
  - If the buffer is not empty (occupied > 0) and is allowed to access the critical section (cs_sem>0)
  - decrements *count*
  - exit *critical section*
  - *Signal processes waiting on due to the full buffer*
  - consumes data item

Consumer-producer using condition variables

```
data_type buffer[N];
int count = 0;
sem_t occupied, empty, cs_sem;
void processA() {
  int i = 0;
  while( 1 ) {
    produce(&data);
    sem_wait (&empty); //decrease empty
    sem_wait (&cs_sem); //decrease cs_sem
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
    sem_post (&cs_sem); //increase cs_sem
    sem_post (&occupied); //increase occupied
  }
}
void processB() {
  int i = 0;
  while( 1 ) {
    sem_wait(&occupied); //decrease occupied
    sem-wait(&cs_sem);   //decrease cs_sem
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    sem_post(&cs_sem);  // increase cs_sem
    sem_post(&empty);   // increase empty
    consume(&data);
  }
}
void main() {
sem_init(&occupied, 0, 0);
sem_init(&empty,0, N);
sem_init(&cs_sem, 0, 1);
create_process(processA); create_process(processB);
}
```

A COMMON PROBLEM IN CONCURRENT PROGRAMMING: DEADLOCK

- Deadlock: A condition where 2 or more processes are blocked waiting for the other to unlock critical sections of code
    - Both processes are then in blocked state
    - Cannot execute unlock operation so will wait forever
- Example code has 2 different critical sections of code that can be accessed simultaneously
    - 2 locks needed (mutex1, mutex2)
    - Following execution sequence produces deadlock
        - *A* executes lock operation on *mutex1* (and acquires it)
        - *B* executes lock operation on *mutex2*(and acquires it)
        - *A/B* both execute in critical sections 1 and 2, respectively
        - *A* executes lock operation on *mutex2*
            - *A* blocked until *B* unlocks *mutex2*
        - *B* executes lock operation on *mutex1*
            - *B* blocked until *A* unlocks *mutex1*
        - DEADLOCK!

```
mutex mutex1, mutex2;
void processA() {
  while( 1 ) {
    …
    mutex1.lock();
    /* critical section 1 */
    mutex2.lock();
    /* critical section 2 */
    mutex2.unlock();
    /* critical section 1 */
    mutex1.unlock();
  }
}
void processB() {
  while( 1 ) {
    …
    mutex2.lock();
    /* critical section 2 */
    mutex1.lock();
    /* critical section 1 */
    mutex1.unlock();
    /* critical section 2 */
    mutex2.unlock();
  }
}
```

MESSAGE PASSING

- Message passing
  - Data explicitly sent from one process to another (*msgsnd, msgget, msgrcv, etc*)
    - Sending process performs special operation, *send*
    - Receiving process must perform special operation, *receive*, to receive the data
    - Both operations must explicitly specify which process it is sending to or receiving from
    - Receive is blocking, sending may or may not be blocking
  - Safer model, overhead can be high
- Two modes:
  - blocking: sending process waits for response;
  - non-blocking: sending process continues.

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

MULTIPLE PROCESS

- Create a process
- Context switching
- Process State and Scheduling
- Interprocess communication
- Real-time operating system (RTOS)

REAL-TIME OPERATING SYSTEMS

- What
  - Operating system with bounded response time
    - Provide mechanisms, primitives, and guidelines for building real-
time embedded systems
- Real-Time
- Operating systems

REAL-TIME SYSTEMS

- Not systems run very fast
- The real-time system is the system that its timeliness is as important as the logic correctness of the result
- Two basic categories
  - Hard real-time
    - Deadline misses imply the failure of system
  - Soft real-time
    - Deadlines can be occasionally missed
    - *Firm real-time system*
      - *Deadline miss is of no use at all*
    - *Non-firm real-time system*
      - *Task execution is still valuable with deadline miss* albeit with reduced *performance*

OPERATING SYSTEMS

- A software that manages system resources and supports user interface to access these resources
  - System resources
    - CPU times
    - Memory usage
    - File handlers
    - Networking
    - Input/output devices
    - etc

- Examples
  - Unix, Linux, Microsoft Windows, Mac OS, etc

CHARACTERISTICS OF RTOS

- Deterministic/Predicability
  - To deliver service in deterministic or predicable time
    - Non-deterministic makes embedded system to randomly miss deadlines, which is not acceptable in real-time systems
  - Scheduling/memory allocation/inter task communication
- Usually small in size
  - Small kernel with optional resource managers

REAL-TIME OPERATING SYSTEMS (RTOS)

- Windows CE
  - Built specifically for embedded systems and appliance market
  - Scalable real-time 32-bit platform
  - Supports Windows API
  - Perfect for systems designed to interface with Internet
  - Preemptive priority scheduling with 256 priority levels per process
  - Kernel is 400 Kbytes
- QNX
  - Real-time microkernel surrounded by optional processes (resource managers) that provide POSIX and UNIX compatibility
    - Microkernels typically support only the most basic services
    - Optional resource managers allow scalability from small ROM-based systems to huge multiprocessor systems connected by various networking and communication technologies
  - Preemptive process scheduling using FIFO, round-robin, adaptive, or priority-driven scheduling
  - 32 priority levels per process
  - Microkernel < 10 Kbytes and complies with POSIX real-time standard

SUMMARY

- Process/thread, reentrancy
- Process/thread creation
- Multitasking context switching
  - Co-operative multitasking
  - Preemptive multitasking
  - Co-routine
- Scheduling
  - Cyclic scheduling
  - Round robin
  - Priority-based preemptive scheduling
    - RMA/EDF
- Interprocess communication
  - Shared memory/message passing
  - Mutex/semaphore
  - Priority inversion/deadlock
- Real-time and Real-time Operating System (RTOS)