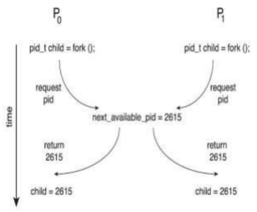
Systems and Devices 2 (SYS2) Module
Threads/Tasks Synchronization
Autumn Term 2021

***Concurrency:***

- Concurrency in OS is allowing more than one process to make progress at a time
- The basic requirement for support of concurrent processes is the ability to enforce mutual exclusion
- Example
    - Processes $P_0$ and $P_1$ are creating child processes using the fork() system call
    - Race condition on kernel variable next_available_pid which represents the next available process identifier, pidfi
    - Unless there is a mechanism to prevent $P_0$ and $P_1$ from accessing the variable next_available_pid the same pid could be assigned to two different processes

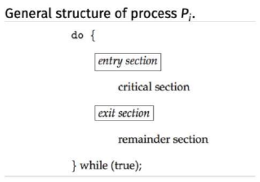


***Race condition:***

- A race condition occurs when multiple processes or threads read and write data items so that the final result depends om the order of execution of instructions in the multiple processes

***Critical Section Problem:***

- Consider system of n processes $p_0$, $p_1$, … $p_{n-1}$
- Each process has critical section segment of code
- Process may be changing common variables, updating table, writing file
- When one process in critical section, no other may be in its critical section, no other may be in the critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then reminder sectionfi


General structure of process $P_i$.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

- Critical section problem solution requirements
    - Mutual exclusion
        - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
    - Progress
        - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
    - Bounded waiting
        - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has mace the request to enter its critical section and before that request is granted
            - Assume that each process executes at a Nonzero speed
            - No assumption concerning relative speed of the n processes

***Requirements for Mutual Exclusion:***

- Mutual exclusion must be enforcedﬁ Only one process at a time is allowed into its critical section, among all processes that have critical section among all processes that have critical sections for the same resource or shared object
- A process that halts in its noncritical section must do so without interfering with other processes
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or starvation
- When no process is in a critical section any process that requests entry to its critical section must be permitted to enter without delay
- No assumptions are made about relative process speeds or number of processors
- A process remains inside its critical section for a finite time onlyﬁ

***Implementation of Mutual Exclusion:***

- One approach is to leave the responsibility with the processes that wish to execute concurrentlyﬁ Processes, whether they are system programs or application programs, would be required to coordinate with one another to enforce mutual exclusion, we can refer to these as software approachesﬁ Although this approach is prone to high processing overhead and bugs, it is nerveless useful to examine such approaches to gain a better understanding of the complexity of concurrent processingﬁ
- A second approach involves the use of special-purpose machine instructionsﬁ These have the advantage of reducing overhead but nerveless will be shown to be unattractive as a general-purpose solution
- A third approach is to provide some level of support within the OS as a programming languageﬁ

***Mutual exclusion – hardware support:***

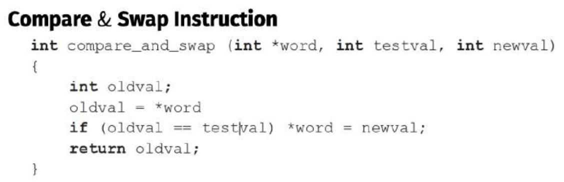
- Atomic operation

- A function or action implemented as a sequence of one or more instructions that appears to be indivisible, that is, no other process can see an intermediate state or interrupt the operationfi The sequence of instruction is guaranteed to execute as a group, r not execute at all, having no visible effect on system statefi Atomicity guarantees isolation from concurrent processes
  - Interrupt disabling
    - In a uniprocessor system, concurrent processes cannot have overlapped execution, they can only be interleavedfi Furthermore, a process will continue to run until it invokes an OS service or until it is interruptedfi This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interruptsfi

***Special Machine Instructions:***

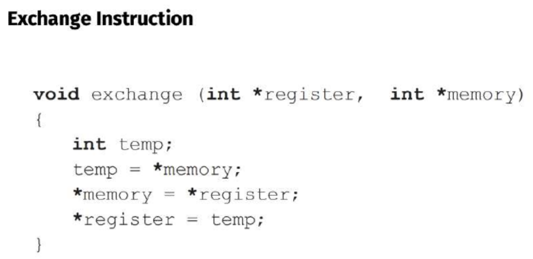
- In a multiprocessor configuration, several processors share access to a common main memoryfi At the hardware level, access to a memory location excludes any other access to that same location
- With this as a foundation, processor designers have proposed several machine instructions that carry out two actions atomically, such as reading and writing or reading and testing, of a single memory location with one instruction fetch cyclefi During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location

***Special Machine Instructions:***



**Compare & Swap Instruction**

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```



**Exchange Instruction**

```
void exchange (int *register,  int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

***Advantages of Machine-instruction approach:***

- It is applicable to any member of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections, each critical section can be defined by its own variablefi

***Disadvantages of Machine-Instruction Approach:***

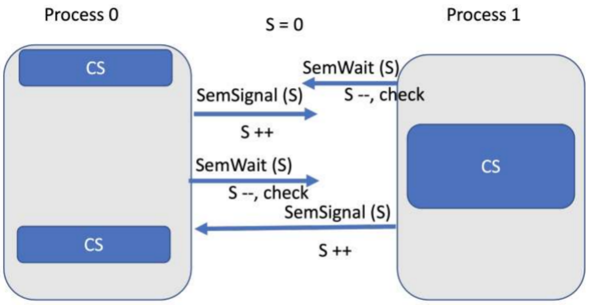
- Busy waiting is employed
  - While a process is waiting for access toa critical section, it continues to consume processor time
- Starvation is possible
  - When a process leaves a critical selection and more than one process is waiting process is arbitraryfi Some process could indefinitely be denied access
- Deadlock is possible
  - Consider the following scenario on a single-processor systemfi Process P1 executes the special instruction (efigfi, compare and swap, exchange) and enters its critical sectionfi P1 is then interrupted to give the processor to P2, which has higher priorityfi If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanismfi Thus, it will go into a busy waiting loopfi However, P1 will never be dispatched because it is of lower priority than another ready process, P2

***OS and Programming language supported concurrency:***

- The idea that two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signalfi Any complex coordination requirement can be satisfied by the appropriate structure of signalsfi Different data structures that can be used are, Semaphore
- Binary Semaphore, Mutex, Conditional Variables, Event Flags, Mailbox messages, spinlocks

***Semaphore:***

- An integer value used for signalling among processes
- Only three operations may be performed on a semaphore, all of which are atomictt initialize, decrement, and increment
- The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a processfi Also known as a counting semaphore or a general semaphorefi
- Example

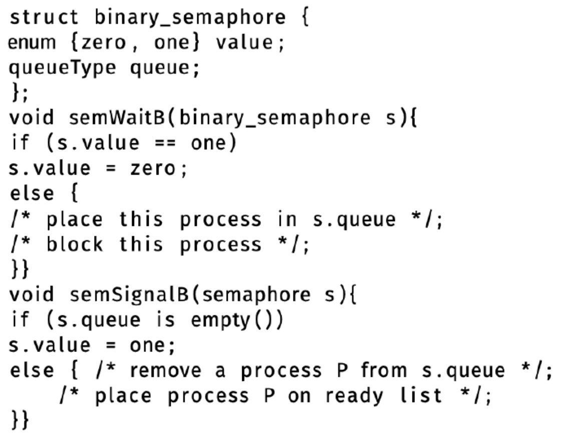


```
struct semaphore {
int count;
queueType queue;
};
void semWait(semaphore s){
s.count--;
if (s.count < o) {
/* place this process in s.queue */;
/* block this process */;
}}

void semSignal(semaphore s){
s.count++;
if (s.count <= o) {
/* remove a process P from s.queue */;
/* place process P on ready list */;
}}
```

***Binary semaphore:***

- A binary semaphore may be initialized to 0 or 1
- The semWaitB operation checks the semaphore valuefi If the value is zero, then the process executing the semWaitB is blockedfi If the value os one, then the value is changed to zero and the process continues execution
- The semiSignalB operation checks to see if any processes are blocked on this semaphore, value equals 0fi If this is the case, then a process blocked by a semWaitB operation unblockedfi If no processes are blocked then the value of the semaphore is set to onefi



```
struct binary_semaphore {
enum {zero, one} value;
queueType queue;
};
void semWaitB(binary_semaphore s){
if (s.value == one)
s.value = zero;
else {
/* place this process in s.queue */;
/* block this process */;
}}
void semSignalB(semaphore s){
if (s.queue is empty())
s.value = one;
else { /* remove a process P from s.queue */;
    /* place process P on ready list */;
}}
```

***Mutex:***

- Similar to a binary semaphore
- A key difference between the two is that the process that locks the mutex, sets the value to 0, must be the one to unlock it, sets the value to onefi

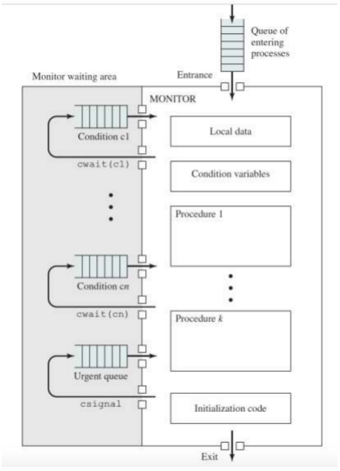
***Monitor and conditional variables:***

- A monitor is a software module consisting of one or more procedures, an initialization sequence, and local datafi The chief characteristics of a monitor are the following
  - The local data variables are accessible only by the monitors procedures and not by any external procedure
  - A process enters the monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time, any other processes that have invoking one of its procedures
  - Only one process may be executing In the monitor at a time, any other processes that have invoked the monitor are blocked, waiting for the monitor to become availablefi

***Condition variables:***

- Condition variables are a special data type in monitors, which are operated on by two functions
  - Cwait
    - Suspend the execution of the calling process on condition cfi The monitor is now available for use by another process
  - Csignal

- Resume execution of some process blocked after a cwait on the same conditionfi If there are several such processes, chose one of the, if there is no such process, do nothing


Queue of entering processes
Entrance
Monitor waiting area
MONITOR
Condition c1
cwait(c1)
Local data
Condition variables
Procedure 1
Condition cn
cwait(cn)
Procedure k
Urgent queue
csignal
Initialization code
Exit