Assignment #2

Saturday, March 8, 2025 11:41 AM

Chapter 2 (30pts, 5pts each)

1. What are the differences between processes and threads?

A process is an instance of a program in execution, with its own isolated memory space, including code, data, and resources (like open files). Each process includes at least one thread, which is the smallest unit of execution. A single process can contain multiple threads.

Threads within the same process share memory (code, data, and heap) but maintain their own stacks and registers, making them lightweight and efficient for communication. Inter-thread communication within a process occurs through shared memory without kernel intervention. In contrast, inter-process communication (IPC) between separate processes requires kernel support and is less efficient.

Additionally, creating and switching between threads is inexpensive compared to processes because threads share resources, whereas creating and switching between processes involves significant overhead and kernel-level support.

2. With a five-state process model, please explain each state and the transitions.

1) **New** - a process has been created but has not yet been admitted into the pool of processes that are ready to run.
2) **Ready** - now the process is waiting to get some time on the CPU. It has everything it needs and is ready to be processed.
3) **Running** - the process has control of the CPU and is currently execting instructions
4) **Blocked** - the process has been stopped from using the CPU and is waiting for an event or other resources to continue (like /O completion)
5) **Terminated** - the process has finished executing its instructions or it has been terminated by other means and is no longer active.

The transistions are as follows:

- **New to Ready** - the OS admits the process into the ready state
- **Ready to Running** - the scheduler sees that the process is ready to go and gives it some time on the CPU.
- **Running to Ready** - the process may be interupted or preempted and is now waiting to be recheulded for CPU time.
- **Running to Blocked** - the process intiates an I/O request or must wait for an event and cannot continue until this even happens.
- **Blocked to Ready** - the event or I/O happens and now the process is ready to get sheduled for CPU time again
- **Running to Terminated** - the proces is now complete or has been killed by other means, but the proces is no longer running.

3. Please discuss the advantages/disadvantages of using user-level and kernel-level threads.

Advantages:
User-Level Threads:
User-level threads allow fast and inexpensive creation and context switching because they are lightweight. Scheduling is managed entirely in user space, and there is no kernel involvement in thread management, which minimizes overhead when switching.

Kernel-Level Threads:
Kernel-level threads are visible to the kernel, so the kernel can schedule each thread independently. This capability allows true parallel execution on multiple CPUs, and a

blocking call in one thread does not block other threads within the same process.

Disadvantages:
User-Level Threads:
Because the kernel sees only one thread per process, if one thread makes a blocking system call, the entire process may be blocked. Additionally, there is limited multiprocessor utilization since the kernel cannot schedule individual threads across processors.

Kernel-Level Threads:
Thread creation and context switching incur higher overhead due to the required system calls and kernel management. They also use more system resources because of the complexity of kernel-level scheduling.

4. What are the advantages/disadvantages of busy-waiting and sleep-and-wakeup approaches for mutual exclusion?

Advantages:
Busy Waiting:
Busy waiting allows a process to enter its critical section without incurring the cost of a context switch when the wait is very short. This approach is efficient when the critical region is brief, as the process simply polls a condition and proceeds immediately when the resource becomes available.

Sleep-and-Wakeup:
The sleep-and-wakeup method conserves CPU cycles because a process that cannot enter its critical region is put to sleep until a wakeup signal is received. This avoids the continuous polling characteristic of busy waiting, thereby reducing unnecessary CPU usage.

Disadvantages:
Busy Waiting:
Busy waiting wastes CPU cycles since the process continuously checks a condition until it becomes true. This inefficiency is especially problematic on single-core systems, where busy waiting can prevent other processes from running.

Sleep-and-Wakeup:
While sleep-and-wakeup avoids wasting CPU cycles, it introduces overhead due to the context switches required to put a process to sleep and then to wake it up. It is important to correctly handle wakeup signals, as any mismanagement may result in lost signals, thereby delaying the process's progress.

5. Why are semaphores needed? What are the similarities and differences between a semaphore and a mutex?

Semaphores are used to control access to shared resources and to prevent race conditions by allowing processes to block when a resource is unavailable and to be awakened when it becomes available. They provide a mechanism to ensure that critical sections are executed safely without interference.

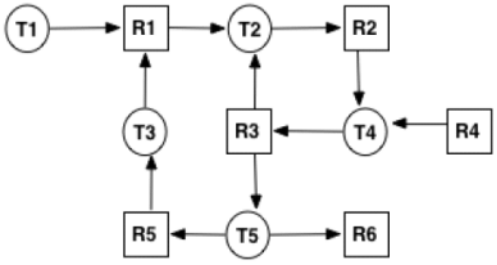
Similarities:
Both semaphores and mutexes are employed to protect critical sections and enforce mutual exclusion when accessing shared resources.

Differences:
Semaphores are a more general synchronization tool. They can be implemented as counting semaphores, which allow a fixed number of processes to access a resource concurrently, or as binary semaphores, which function similarly to mutexes. In contrast, a mutex is specifically designed for mutual exclusion and typically permits only one thread to enter a critical section at a time. Moreover, semaphores lack ownership, meaning any process or thread can signal (release) them, while a mutex is owned by the thread that locks it, and only that same thread should unlock it.

6. In the dining philosophers problem's solution (Section 6.1.3, Figure 6-5, or Lec7 slide 43-44), explain briefly what does the function `test(i)` do (two roles) and how it works.

The function test(i) has two main roles in the dining philosophers solution:
1) It first checks if philosopher i is in the hungry state and if neither of its immediate neighbors is eating. If both conditions are met, it changes philosopher i's state to eating and performs an up() on its semaphore to allow the philosopher to proceed.

2) It is also used after a philosopher finishes eating (in the put_forks routine) to test whether the left or right neighbor, which might be waiting to eat, can now start eating. This call helps to unblock a neighbor when the forks become available.



(a) Chapter 6, Question 1.

	C matrix				R matrix			
P1	0	0	1	1	2	0	0	0
P2	0	1	0	1	0	0	1	1
P3	0	1	1	0	1	1	0	2
P4	1	0	1	0	1	2	1	0
P5	0	0	0	1	2	2	0	3

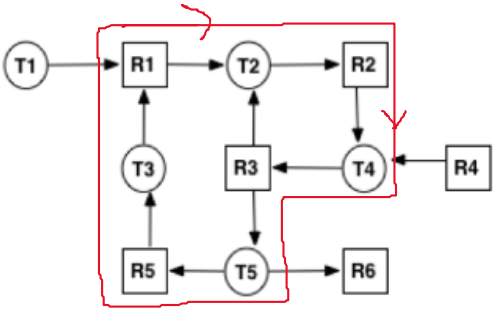
(b) Chapter 6, Question 2.

Figure 1: Chapter 6 questions.

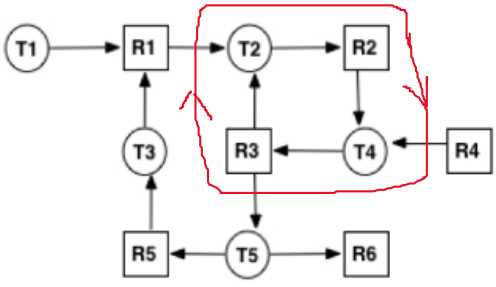
Chapter 6 (20pts)

1. Show how the deadlock detection algorithm works on the resource graph in Figure 1a (previous page) to find all potential deadlocks. **Please show your work, and identify the processes and resources involved** (8 pts).

L1 = [R1, T2, R2, T4, R3, T5, R5, T3, R1] = Deadlock
L2 = [T4, R3, T2, R2, T4] = Deadlock



L1



L2

2. A system has four resources and five processes. The resources in existence vector is E = (2 2 4 4). The current allocation (C matrix) and requests (R matrix) are shown in Figure 1b (previous page). What is the resources available vector A? Can you detect any deadlock? **Please show your work** (12 pts).

	C matrix				R matrix			
P1	0	0	1	1	2	0	0	0
P2	0	1	0	1	0	0	1	1
P3	0	1	1	0	1	1	0	2
P4	1	0	1	0	1	2	1	0
P5	0	0	0	1	2	2	0	3

E = 2244
P = 1233 (add up the cols of C Matrix)
A = 1011 (A = E - P)

R[2] <= A so A = (1011) + C[2] (0101) = (1112)
R[3] <= A so A = (1112) + C[3] (0110) = (1222)
R[4] <= A so A = (1222) + C[4] (1010) = (2232)
R[1] <= A so A = (2232) + C[1] (0011) = (2243)
R[5] <= A so A = (2243) + C[5] (0001) = (2244)

A = E so no dealock